Translation of Serial Recursive Codes to Parallel SIMD Codes

Abdou Youssef
Department of EE & CS
The George Washington University
Washington, DC 20052
USA
email: youssef@seas.gwu.edu

Abstract

Parallelizing compilers are an important way for making parallel systems easy to use and more acceptable to the mainstream of computing. Several approaches to parallelization have been taken, such as loop parallelization and dependence-graph based parallelization. This paper will undertake another approach: the exploitation of recursion in serial recursive coders to translate the latter to parallel non-recursive SIMD codes. The principal idea is to perform first a formal data-dependence data-flow analysis on both the input partitioning process and the subsolution-merge process to determine the computation and communication of each partition and each merge, then develop a system of equations whereby we derive the sequences of computation instructions and communication instructions along with their processors-enabling/disabling masks for the SIMD target code. We develop the approach, give a code translation algorithm, and apply it to a sample of common recursive algorithms to illustrate its power. The approach is very effective and efficient, especially for recursive algorithms that are balanced and whose input can be padded to an appropriate size without affecting the desired output. For some recursive algorithms that do not fit this category, it will be argued that a top-down MIMD execution of these algorithms is preferable to SIMD execution. Finally, we note that our approach is generalizable to interleaved recursion.

1 Introduction

Parallelism is a powerful approach to high-performance computing, and great advances have been made in building and programming parallel systems. Despite these advances, however, parallelism has not entered the general mainstream of computing. One important way to greatly improve this situation is to have parallelizing compilers, which allow users to continue to program in the conventional (i.e., serial) manner, while still benefiting from the speedup of parallel machines. Various approaches to parallelizing compilers have been taken, such as loop parallelization [1, 11] and dependence graph based parallelism [2, 10]. Another fruitful approach, which will be undertaken in this paper, is the exploitation of recursion.

Recursion has proved to be a highly useful programming mechanism. It is a natural and effort-saving way of designing and expressing algorithms based on powerful algorithmic paradigms, e.g., divide and conquer. Interestingly, recursion is very suitable for parallel systems without requiring any change or additional effort on the part of programmers. In the top-down view, recursion yields to control-level parallelism by spawning a new process for each recursive call and assigning it to a new processor. This is suitable for multiple-instruction, multiple-data machines (MIMD). In the bottom-up view of recursion, where the recursive structure is unfolded and converted into an iterative tree-like structure, there is usually a considerable amount of data parallelism, i.e., in each level of the tree the various computational nodes can run simultaneously on different processors. This matches data-parallel machines, often implemented as single-instruction, multiple-data machines (SIMD). Since data parallelism incurs much less control and synchronization overhead than control-level parallelism, we will pursue the bottom-up view of recursion to convert (serial) recursive codes to data-parallel non-recursive SIMD codes.

In the SIMD model, a program consists of a sequence of instructions, very much like serial programs for serial machines, with two major differences. First, every SIMD instruction is broadcast to all processors and then executed by the enabled processors, each on its own local data. Second, more than computation instructions are needed. In fact, for the proper execution of a SIMD program, the data has to be appropriately laid out and distributed to the individual processors at the outset, and communication instructions must be provided so that enabled processors can exchange data when needed. Also, a mechanism is required to enable/disable processors. Therefore, the SIMD compiler envisioned here converts a recursive serial code into a non-recursive SIMD sequence of instructions comprising four categories: *data layout instructions*, *computation instructions, communication instructions*, and *processor enabling/disabling (E/D) instructions* (i.e., if-then instructions or masks).

Our code generation approach is general in the sense that it applies to any recursive code, however structured, and in many cases the generated SIMD codes are optimal with respect to speed and efficiency. For example, in well-balanced recursive codes, as in two-way recursive codes where each of the two recursive calls operates on half of the original input data, the generated SIMD code is optimal. For not so balanced recursive codes, the generated SIMD code may

not be optimal; in fact, we will point out later which kind of recursive serial codes is best translated into SIMD code (using our approach), and which kind is better executed in MIMD mode. Another current limitation is that interleaved recursion is not handled here. This may not be terribly restrictive, however, since most recursive codes are not interleaved. Also, the approach may be extended in the future to handle interleaved recursion.

The paper is organized as follows. In the next section we overview the SIMD model. Section 3 develops the code generation algorithm, which converts a serial recursive code into a sequence of SIMD instructions. That section also carries out an analysis of when a SIMD target code for a recursive source code is good and when a MIMD target is preferable. Section 4 illustrates the working of the code generation algorithm by applying it to two representative recursive algorithms, namely, semi-group computations, and FFT [5]. Finally, concluding remarks and future work are presented in Section 5.

2 The SIMD Model

In this section we overview the relevant elements of the SIMD model, particularly the aspects of computation, communication, and processor enabling/disabling mechanisms.

A SIMD system consists of a single control unit (CU), a number of identical processors, a global broadcast bus connecting the control unit to all the processors to broadcast instructions and data from the CU, an interconnection network over which the processors can communicate with one another, and an input/output system connected to the CU and possibly to some nodes in the network.

A SIMD program is a sequence of instructions and resides in the control unit. The instructions are IO instructions, computation instructions, or communication instructions. The execution of a computation instruction is carried out as follows. If the instruction is a single, serial operation, then it is executed locally in the control unit. On the other hand, if the instruction is an operation to be applied uniformly on many data elements, such as the addition of two vectors, then the instruction is broadcast to all the processors, and some (or all) of the processors, the enabled ones, execute the instruction on their local data. The mechanism to enable/disable processors is explained later in this section.

If the instruction is a communication instruction, it is broadcast to all the processors to be executed by the enabled ones. Typically, the SIMD communication instructions can have one of two possible forms. The first form involves having a set of single-step basic communication instructions in terms of which any communication pattern is expressed. For example, in a mesh network there can be 4 basic communication instructions: *north, south, east,* and *west.* A north-instruction broadcast to the processors causes each enabled processor to send the data item in its local output register to its north neighbor. In the n-cube, as another example, there can be n basic communication instructions (e_i), for $i = 0, 1, \ldots, n-1$, where $e_i \equiv$ "send along dimension i", that is, the processors whose binary ID's differ only in the i-th bit exchange data. As will be seen, many of the widely used recursive algorithms are best implemented using these n-cube basic communication steps (e_i). It's worthy to note that besides the n-cube several other networks support the (e_i)'s, most notably the Omega network [8] and the Benes/Clos network [4].

The second form of communication instructions involves a stronger abstraction of the communication process, decoupling the operation of the network from the operation of the processors. In this form, each processor has, among other things, one (or more) pair of registers: an output register (the processor's gateway to the network) and an address register containing the ID of the destination processor. In this case, only one communication instruction is needed, *send*, which is broadcast to all the processors when needed, and every enabled processor executes it by passing to the network the data in its output register(s) and the address(es) in its address register(s). The network then independently makes sure that the data is delivered to the destination processors. This form clearly frees the compiler designer from worrying about communication in its lowest hardware level, thus allowing for portable software.

Before proceeding to other types of instructions, note that special data movement instructions are used to move data from local memory to local registers in each processor prior to computation or communication operations. Data movement can be done in a standard way like in any SIMD system. Therefore, for our purposes, we need not concern ourselves with this local data movement, to keep the presentation clear and focused, without compromising much the completeness of the code generation process.

The instructions for IO and for data layout are a special form of communication instructions. Therefore, although IO is critical to performance, we need not treat it separately in our code translation, but address it as communication.

Finally, a SIMD system must have an enable/disable (E/D) mechanism. There are at least two E/D mechanisms. The first is through if-then control instructions executed by all the processors to determine which processors are to execute the next instruction or block of instructions. For example, "if processor-ID is even, then enable, else disable" clearly enables all even-numbered processors and disables the others. The second approach is to associate a *mask* with each instruction, where the processors that "fit" the mask are enabled and execute the corresponding instruction. The mask mechanism, though less general, is clearly faster than the first mechanism if additional bus wires are available for the mask bits, since the E/D is done on the fly simultaneously with the execution of the instructions. Although our code generation algorithm can work with either or both types of E/D mechanisms, we will mostly focus on the mask mechanism because of its elegance and speed.

A mask is simply a tertiary string over the alphabet $\{0, 1, *\}$, where a star signifies a "don't care". For example, $0**0$, $1*0*$, 0110, and $****$ are four masks. The non-star symbols of a mask are called *fixed bits.* A processor ID, which is a binary string, is said to fit a given mask if the fixed bits of the mask are identical to the corresponding bits of the processor ID. In the mask E/D mechanism, the processors that fit the mask are enabled, while all other processors are disabled, until a new mask is broadcast. For instance, the all-star mask enables all processors, an all-fixed-bits mask enables exactly one processor (whose ID is the mask itself), and $0**0$ enables processors $\{0000, 0010, 0100, 0110\}$. The power and limitations of this masking scheme should be clear.

Before closing this section, we will give the notation for two operations that will be used in the target code generation: string concatenation (for masks), and instruction-mask joining. The concatenation $x \bullet y$ of two strings $x = a_1 \ldots a_n$

```
PRM(input I)
begin
 if input size is small enough then
    do the necessary work; {Comment: the basis step}
    return;
 endif

 Partition the input into n parts I_1, I_2, ..., I_n;
    {Comment: the input partitioning can be a
    simple subdivision of the input into contigu-
    ous parts, often of equal size, or an elaborate
    processing of the input before invoking recur-
    sion. }

 Call PRM(I_1) to get subsolution S_1;
 Call PRM(I_2) to get subsolution S_2;
      .
      .
      .
 Call PRM(I_n) to get subsolution S_n;
    {n is most often equal to 2. In any case,
    n will be assumed here to be a power of 2.}

 Merge subsolutions S_1, ..., S_n into one solution S;
end
```

Figure 1: **The Recursive Source Code Template to Be Considered**

and $y = b_1 \dots b_m$ is the string $xy = a_1 \dots a_n b_1 \dots b_m$. The join of an instruction i and a mask m, denoted $i \oplus m$, is a convenient way to designate a SIMD instruction i that will be executed by the processors that fit the mask m. If I is a sequence of q instructions, and M is a sequence of q masks, then $I \odot M$ is the sequence of q pointwise joins of the respective elements.

3 The Recursive-Serial to SIMD Code Translation

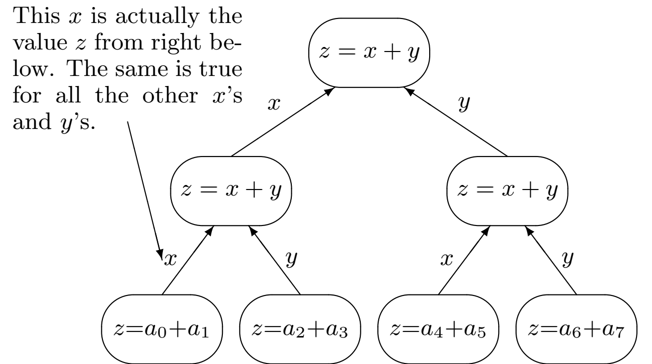
The principal idea of the translation process is to perform first a formal data-dependence data-flow analysis on both the input partitioning process and the subsolution merge process, and then conclude the sequences of computation instructions and communication instructions along with their E/D masks for the SIMD target code.

In principle, our translation approach can translate any non-interleaved recursive code. However, we will apply the approach to just one recursive structure, which we term *the n-way partition-recur-merge (PRM)* structure; a template of the n-way PRM is presented in Figure 1, and a simple example (addition of N numbers) is shown in Figure 2. We choose this structure for several reasons. First, it captures most of the common and important recursive algorithms that are designed using divide and conquer. Second, by restricting ourselves to a clean structure, we simplify the presentation and highlight better the main ideas of our approach. Finally, as will become apparent later, for more involved recursive structures, and even for the same structure but with unbalanced input size (e.g., not a power of 2), the generated SIMD code may take significantly longer to execute than a MIMD counterpart.

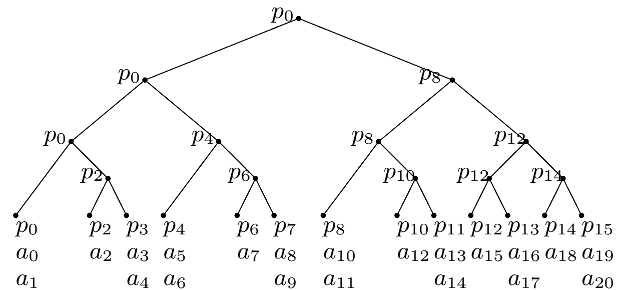
```
function Add(a(i..j))
begin
  if i = j    { input size equal to 1}
     z = a(i); return (z);
  elseif j = i + 1      { input size equal to 2}
     z = a(i) + a(i + 1);
     return (z);
  endif

  { Now the data size is > 2, and recursion is used}
  x = Add(a(i..(i+j)/2));
  y = Add(a((i+j)/2 + 1..j));
  z = x + y;
  return (z);
end
```

Figure 2: **A Recursive Code for the Addition of** N (= $j - i + 1$) **Numbers**



– The full tree of the Add code for input $a_0, \dots, a_7$



– The non-full tree of Add-ing 21 numbers $a_0, \dots, a_{20}$.
– The p_i's are the ID's of the host processors.
– Below each leaf is 1 or 2 input data for the basis step .

Figure 3: **A Full Tree and a Non-Full Tree of the Add Code**

To develop the code translation algorithm, we observe first that if the recursion in the n-way PRM source code is unfolded, we obtain a tree with the following features (the reader may refer to Figure 3 for illustrations):

- The root has n children, and the i-th subtree of the root corresponds to the i-th recursive call. Also, each node in the tree has at most n children.
- Each leaf node is an instance of the basis step of the source code, executed on a piece of the original input (that may possibly have been modified as part of the partitioning process) of size at most σ, where σ is the maximum allowed input size of the basis step in the source code..
- The tree is executed top-down first (for input partitioning), and then bottom-up for merging. In top-down, each internal node is an instance of the partition step, partitioning its input into parts that are passed to the children nodes. In bottom-up, each internal node is an instance of the merge step merging the outcomes (i.e., subsolutions) of its children nodes.
- All the nodes (computations) in any one level of the tree are independent and identical computations (but performed on different data). Therefore, they are suitable for parallel SIMD execution, where each node of the level runs on a separate set of (one or more) processors. Once a level is complete, some communication takes place, and then the next level can start. Note that we number the levels in an upward data-dependence fashion: the leaves are in level 0, all the nodes that can run after the leaves are in level 1, and so on up to level l of the root, for some l.
- Based on the last three observations, the target SIMD code to be generated has the format shown in Figure 4. The code translation must then focus on determining the exact computation and communication instructions — along with their associated masks (see below) — in the main for-loops of Figure 4.
- The number of nodes varies from level to level. Therefore, not all the processors are always needed in each level, and E/D masking has to be used.
- The communication needed between successive levels of computation is between the processors hosting a parent node and the processors hosting its children. The choice of the host processors of a parent node affects both the pattern of communication and the masking needed for the communication and computation. The number and choice of the host processors will be determined by symbolic analysis of the partition step and the merge step of the source code.
- The exact shape of the tree from the same n-way PRM source code varies depending on the input size (see Figure 3). Thus, the communication and masks are data-size dependent. This complicates the code generation process considerably.

In addressing the difficulty arising from data-size dependent trees, we distinguish three important cases:

1. The tree is a full n-ary tree. Examples include addition of 2^K numbers, FFT of 2^K numbers [5], bitonic sorting of 2^K elements [3], and certain inherently power-of-two transforms in digital signal processing [6] such as the Haar transform, the Walsh transform, and the Hadamard transform. This case will be handled satisfactorily.
2. The tree is not full but the input I of the source code can be padded by a certain number of a certain value so that (1) the resulting tree of the new input I' is full, and (2) the desired output corresponding to I is a well-defined zone of the output corresponding to input I'. This case covers most of the common and important recursive algorithms. Since this case reduces to the previous case, it will be handled effectively.
3. The tree is not full and the input I cannot be padded as in the previous case. In this case, there is no way around run-time mask generation and run-time identification of the communication patterns. As will be seen later, under this case the execution of the tree in a top-down MIMD way, where a new process is spawned for every recursive call, may often be a preferable way of execution.

We now handle each case separately.

3.1 Case 1: Full Tree

To streamline the discussion, we assume first that the input partitioning is a straightforward contiguous subdivision of the input I into equal parts, and focus on merging. Later, involved partitioning will be easily addressed in a very similar fashion — partitioning is the reverse process of merging and yields to the same treatment as merging.

In the case of full tree, the number of leaves in the tree is n^l for some l. This arises when the input size $N = \sigma \times n^l$ (e.g., in the addition of N numbers, $\sigma = 2$ and $n = 2$, leading to a power-of-2 input size N).

Layout

The layout under the assumption of the contiguous subdivision of the data into equal parts is straightforwad. First, since there are n^l leaves, we use n^l processors, logically labeled $0, 1, \ldots, n^l - 1$. Second, we subdivide the input I into n^l contiguous parts R_i of size σ each. Third, we assign part R_i to processor i, for all $i = 0, 1, \ldots, n^l - 1$. We call this layout *the canonical layout.*

The computation/communication sequences of instructions and their masks

The instructions of the basis step of the source code are executed by all the n^l processors, each processor i on its local data R_i. The mask for each of these instructions is simply the all-star mask, that is, all the processors are enabled.

In the remaining levels of the tree, each node is a merge node merging the subsolutions from its n children. This subpattern of the tree is very critical, and will be addressed in a symbolic formal manner. We term it the *merge-kernel*($\mathcal{N}$), where $\mathcal{N}$ is the size of the input of the merge node (see Figure 5-(b)). For example, the $\mathcal{N}$ of the root merge node is n^l. The analysis of the merge kernel will yield all the computation and communication instruction sequences and their corresponding masks for the SIMD target code.

SIMDcode(**input** I)
begin
1- **for** $k = l$ **down to** 1 **do**
– Perform the partition at level k of the tree: every enabled processor performs the partition-computation on the data it has just received from its parent node; some intermittent communication between the processors hosting the node may be needed to complete the partition-computation at this level;

{**Comment**: Denote by $Down.CP_k$ the sequence of computation instructions needed here, and by $Down.M_k^{CP}$ the associated mask sequence. Also, denote by $Intra.part.CM_k$ the communication instructions required here, and by $Intra.part.M_k^{CM}$ the associated mask sequence.}

– Perform the partition-communication from level k to level $k-1$ to send the subinputs to the processors hosting the children nodes;

{**Comment**: Denote by $Down.CM_k$ the sequence of partition-communication instructions needed here, and by $Down.M_k^{CM}$ the associated mask sequence. This communication is different from the one above.}
endfor

{**Comment**: Note that if no partition-processing of the input is specified in the source code, i.e., the partitioning is a straightforward subdivision of the input into contiguous parts, then the previous **for**-loop need not be performed. Rather, the layout of the input data is done canonically.}

2- Every leaf-hosting processor does the basis step of the source code on its local data;

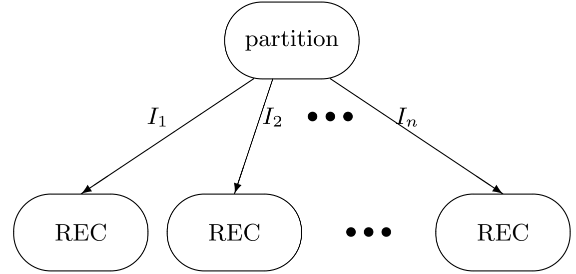
3- **for** $k = 1$ to l **do**
– Perform the merge-communication from level $k-1$ to level k;

{**Comment**: Denote by $Up.CM_k$ the sequence of communication instructions needed here, and by $Up.M_k^{CM}$ the associated mask sequence.}

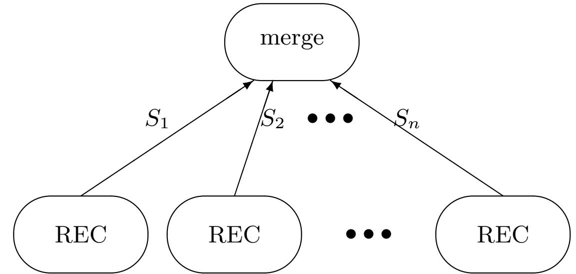
– Perform the merge-computation at level k of the tree: every enabled processor performs the merge-computation on the data it has just received from its children nodes; some intermittent communication between the processors hosting a node may be needed to complete the merge at this level;

{**Comment**: Denote by $Up.CP_k$ the sequence of computation instructions needed here, and by $Up.M_k^{CP}$ the associated mask sequence. Also, denote by $Intra.mrg.CM_k$ the communication instructions required here, and by $Intra.mrg.M_k^{CM}$ the associated mask sequence.}
endfor
end

Figure 4: **The Format of the Target SIMD Code to Be Generated**



The Partition Kernel
(REC ≡ recursive call)
(a)



The Merge Kernel
(b)

Figure 5: **The Partition-Kernel and the Merge-Kernel**

In this pursuit, a symbolic data-dependence analysis is performed on the *merge-kernel*(n^l), where n^l is treated as a formal argument in the analysis. Symbolic analysis is now a mature area [2, 7, 9, 10] and will not be elaborated here. The analysis determines in a simple manner (1) which elements of each subsolution S_i is needed in the computation of new values of the merged solution, and (2) which elements (if any) of each subsolution will be included without change in the merged solution. The next step of the analysis is performed under the following assumptions:

1. The n subsolutions $S_1, S_2, \ldots, S_n$ reside (in a specified way) in n disjoint parts E_i's of the SIMD machine. Similarly, the final solution S is to reside in a certain specified set E of processors. The residing of each S_i relative to its machine part E_i is identical to the way S resides in E.

2. To prevent the requirement of too large a number of processors by the final SIMD target code, we adopt the layout convention that the merge node will be executed by at most as many processors as the combined number of processors needed by its n children. This assumption may lead perhaps to larger granularity as we go up the tree, but for most applications the number of processors needed by each merge node is equal to 1 or to exactly the combined number of processors of the children nodes, thus causing no increase in granularity.

3. To take full advantage of the independence between the subtrees of the recursive-code tree structure, and based on the above, the processors that execute a merge node are taken to be a subset of the processors that execute its children nodes. This limits the communication between two successive levels of the tree to being internal to each machine part that hosts a node and its children. It will also simplify the masking.

4. Since most SIMD systems have a power-of-2 number of processors and are naturally partitionable into submachines of a power-of-2 size, we adopt the convention that each *merge-kernel*(n^l) will reside in a part of the machine of size a power of 2. The merge node of the kernel will be assumed to reside in a power-of-2 subpart.

Based on the outcome of the symbolic data-dependence data-flow analysis, and under the assumptions just stated, the computation/communication steps and their masks can be easily determined in a symbolic form in terms of n^l. Formally, the outcome of this analysis is:

- $CP_{merge}(n^l)$: The computation sequence of the merge step.
- $M^{CP}_{merge}(n^l)$: the mask sequence of the computation sequence $CP_{merge}(n^l)$.
- $Intra.mrg.CM(n^l)$: The intra-communication sequence of the merge step.
- $Intra.mrg.M^{CM}(n^l)$: The mask sequence of the sequence $Intra.mrg.CM(n^l)$.
- $CM_{merge}(n^l)$: The inter-level communication sequence of the merge step.
- $M^{CM}_{merge}(n^l)$: The mask sequence of the communication sequence $CM_{merge}(n^l)$.

REMARK 1:
The communication sequence $Intra.mrg.CM(n^l)$ must be appropriately intertwined with the computation sequence $CP_{merge}(n^l)$ because the merge-computation and the intra merge communication may have to be intermixed for the merging to take place. The way to implement that is to put markers in these two sequences so we know how to intermix the two sequences into one whole. We will not complicate the notation any further; rather, we leave these markers implicit. This is justifiable since the lengths of these sequences do not depend on l, neither do the locations of the markers. (l plays a role in the **contents** of the instructions of these two sequences but not in their length.)

The heart of the SIMD code generation process is to compute the sequences $Up.CP_k$, $Up.CM_k$, $Intra.mrg.CM_k$, $Up.M^{CP}_k$, $Up.M^{CM}_k$, and $Intra.mrg.M^{CM}_k$, i.e., the computation instruction sequence and the 2 communication instruction sequences, along with their masks sequences, for iteration k of the second for-loop of the code SIMDcode (of Figure 4) . The next theorem gives a system of relations that makes the computation of these sequences a straightforward process. (The reader may refer back to Figure 4 to recall the notation.)

Theorem 1 *For $k = 1, 2, \ldots, l$, the following statements hold:*

1. $Up.CP_k = CP_{merge}(n^k)$
2. $Up.M^{CP}_k = *^{(l-k)log_2 n} \bullet M^{CP}_{merge}(n^k)$
3. $Intra.mrg.CM_k = Intra.mrg.CM(n^k)$
4. $Intra.mrg.M^{CM}_k = *^{(l-k)log_2 n} \bullet Intra.mrg.M^{CM}(n^k)$
5. $Up.CM_k = CM^{CM}_{merge}(n^k)$
6. $Up.M^{CM}_k = *^{(l-k)log_2 n} \bullet M^{CM}_{merge}(n^k)$
7. *The locations where $Up.CP_k$ and $Intra.mrg.CM_k$ must intermix are the same as those of $CP_{merge}(n^l)$ and $Intra.mrg.CM(n^l)$.*

Proof: At level k there are n^{l-k} merge nodes (recall that the tree is a full n-ary tree), each being the merge node of a *merge-kernel*(n^k) because the partitioning is simply a subdivision of the input into n equal contiguous parts. Each such kernel runs on a separate part of the SIMD system, and, relative to its machine part, has the sequences $CP_{merge}(n^k)$, $M^{CP}_{merge}(n^k)$, $Intra.mrg.CM(n^k)$, $Intra.mrg.M^{CM}(n^k)$, $CM_{merge}(n^k)$ and $M^{CM}_{merge}(n^k)$. As the kernels have identical computations and communications (on their own local data), then $Up.CP_k$, $Intra.mrg.CM_k$ and $Up.CM_k$ must be simply the same computation/communication of each kernel. Thus statements 1, 3 and 5 of the theorem are proved. The same arguments apply to statement 6; the reason that the locations of the markers do not depend on k (or l) can be found in Remark 1 made earlier. As for the masks, if M is a mask for each kernel relative to its SIMD system part (treated as a standalone SIMD system), then the same mask relative to the whole system is

$ID_{part} \bullet M$, where ID_{part} is the ID of the system part considered. Therefore, the enabled processors for all the kernels at level k are the union of all the enabled processors of all the parts, that is, $\cup_{i=0}^{n^{l-k}-1}(ID_{part\ i} \bullet M)$, which is equal to $(\cup_{i=0}^{n^{l-k}-1} ID_{part\ i}) \bullet M$; by labeling the parts contiguously, i.e., $ID_{part\ i} = i$, $(\cup_{i=0}^{n^{l-k}-1} ID_{part\ i})$ becomes the set $\{0, 1, \ldots, n^{l-k} - 1\}$, which is represented by the mask $*^{\log_2 n^{l-k}} = *^{(l-k)\log_2 n}$. This proves statements 2, 4 and 6 of the theorem. Q.E.D.

Now we return to the case where the input partitioning is more involved, that is, it requires some processing (termed here partition-processing) of the input I before subinputs $(I_i)_{1 \le i \le n}$ are generated and passed to the recursive calls. The same analysis done on the *merge-kernel*(n^l) can and should be done on the *partition-kernel*(n^l), yielding the corresponding communication and computation instruction sequences along with their masks: $CP_{partition}(n^l)$, $Intra.part.CM(n^l)$, $CM_{partition}(n^l)$, $M^{CP}_{partition}(n^l)$, $Intra.part.M^{CM}(n^l)$, and $M^{CM}_{partition}(n^l)$. We stipulate that the initial input layout is such that the partition-processing (i.e., $CP_{partition}(n^l)$) can start before any communication is needed. One exception is when the partitioning is nothing more than reordering of the input. In that case, the initial layout is simply the canonical layout, and the reordering is treated as pure communication. The communication must guarantee that the layout of subinput I_i in its hosting machine part is canonical. Finally, we make the simplifying assumption that the input size remains the same after the partition-processing stage. If this is not the case, some minor changes have to be made to our analysis, but the essential points and arguments are the same.

By doing the same analysis on the *partition-kernel*(n^l) as was done on the *merge-kernel*, a theorem similar to the previous theorem can be derived. We thus present the next theorem without proof.

Theorem 2 *For $k = 1, 2, \ldots, l$, the following statements hold:*

1. $Down.CP_k = CP_{partition}(n^k)$
2. $Down.M^{CP}_k = *^{(l-k)log_2 n} \bullet M^{CP}_{partition}(n^k)$
3. $Intra.part.CM_k = Intra.part.CM(n^k)$
4. $Intra.part.M^{CM}_k = *^{(l-k)log_2 n} \bullet Intra.part.M^{CM}(n^k)$
5. $Down.CM_k = CM^{CM}_{partition}(n^k)$
6. $Down.M^{CM}_k = *^{(l-k)log_2 n} \bullet M^{CM}_{partition}(n^k)$
7. *The locations where $Down.CP_k$ and $Intra.part.CM_k$ must intermix are the same as those of $CP_{partition}(n^l)$ and $Intra.part.CM(n^l)$.*

The SIMD code generation algorithm can now be easily derived. It is shown in Figure 6. The time complexity of this code generation algorithm is clearly dominated by the time to perform symbolic analysis of the merge-kernel and the partition-kernel, which is linear in the number of variables and program statements of the merge step and the partition step of the source code. Consequently, it is a fast algorithm.

SIMD-CODE-GENERATE(**input:** Recursive Source Code)
begin
1- **if** the input partitioning involves processing **then**
– identify *partition-kernel*(n^l) from the source code, perform symbolic analysis on it, and compute the sequences $CP_{partition}(n^l)$, $Intra.part.CM(n^l)$, $CM_{partition}(n^l)$, $M^{CP}_{partition}(n^l)$, $Intra.part.M^{CM}(n^l)$, and $M^{CM}_{partition}(n^l)$;
– for a formal argument k compute
$Down.CP_k = CP_{partition}(n^k)$;
$Down.M^{CP}_k = *^{(l-k)log_2 n} \bullet M^{CP}_{partition}(n^k)$;
$Down.CM_k = CM^{CM}_{partition}(n^k)$;
$Down.M^{CM}_k = *^{(l-k)log_2 n} \bullet M^{CM}_{partition}(n^k)$;
$Intra.part.CM_k = Intra.part.CM(n^k)$;
$Intra.part.M^{CM}_k = *^{(l-k)log_2 n} \bullet Intra.part.M^{CM}(n^k)$;
{Next, join instructions with masks}
$Masked.Down.CP_k = Down.CP_k \odot Down.M^{CP}_k$;
$Masked.Down.CM_k = Down.CM_k \odot Down.M^{CM}_k$;
$Masked.Intra.part.CM_k = Intra.part.CM_k \odot Intra.part.M^{CM}_k$;
endif
2- *merge-kernel*(n^l) analysis:
– identify *merge-kernel*(n^l) from the source code, perform symbolic analysis on it, and compute the sequences $CP_{merge}(n^l)$, $Intra.mrg.CM(n^l)$, $CM_{merge}(n^l)$, $M^{CP}_{merge}(n^l)$, $Intra.mrg.M^{CM}(n^l)$, and $M^{CM}_{merge}(n^l)$;
3- for a formal argument k compute
$Up.CP_k = CP_{merge}(n^k)$;
$Up.M^{CP}_k = *^{(l-k)log_2 n} \bullet M^{CP}_{merge}(n^k)$;
$Up.CM_k = CM^{CM}_{merge}(n^k)$;
$Up.M^{CM}_k = *^{(l-k)log_2 n} \bullet M^{CM}_{merge}(n^k)$;
$Intra.mrg.CM_k = Intra.mrg.CM(n^k)$;
$Intra.mrg.M^{CM}_k = *^{(l-k)log_2 n} \bullet Intra.mrg.M^{CM}(n^k)$;
{ Next, join instructions with their masks}
$Masked.Up.CP_k = Up.CP_k \odot Up.M^{CP}_k$;
$Masked.Up.CM_k = Up.CM_k \odot Up.M^{CM}_k$;
$Masked.Intra.mrg.CM_k = Intra.mrg.CM_k \odot Intra.mrg.M^{CM}_k$;
4- Generate the following SIMD code:
SIMDcode(I) {Size of I is $\sigma \times n^l$}
begin
for $k = l$ **down to** 1 **do**
{This is for input partitioning and data layout}
$Masked.Down.CP_k$ intertwined-with $Masked.Intra.part.M^{CP}_k$;
$Masked.Down.CM_k$;
endfor
The basis step; {By all the processors}
for $k = 1$ **to** l **do** {for upward merging}
$Masked.Up.CM_k$;
$Masked.Up.CP_k$ intertwined-with $Masked.Intra.mrg.M^{CP}_k$;
endfor
end
end

Figure 6: **The SIMD-Code-Generation Algorithm for Full-Tree n-Way PRM**

3.2 Case 2: Input Paddable to the Desired Size

Recall that for an algorithm (or a problem) to fall under this case, its input must be extendable to a desired size (e.g., power of 2) and the solution corresponding to the original input must be easily recoverable from the solution corresponding to the extended input. Fortunately, many interesting and widely used algorithms have this property. All semi-group computations where the associative operator has an identity element, prefix computations, the Discrete Fourier Transform (DFT), and sorting, are just a few examples.

A semi-group computation is the computation of $a[0] * a[1] * \cdots * a[N-1]$ for a given input array a, where $*$ is an associative binary operator. If $*$ has an identity element e (i.e., $x * e = e * x = x$ for all x), then any input array a is extendable to a power-of-two size by padding it with a certain number of e's. The output of the extended input is clearly identical to the output of the original input. Examples of such binary operators include addition, multiplication, minimum, maximum, Boolean "and", Boolean "or", and "exclusive or", with identity elements 0, 1, $+\infty$, $-\infty$, 1, 0, and 0, respectively.

A prefix computation is the computation of an array $A[0..N-1]$ from an input array $a[0..N-1]$: $A[k] = a[0] * a[1] * \cdots * a[k]$, where $*$ is an associative binary operation. Here again, if $*$ has an identity element e, and if N is not a power of two, we can pad the input a with enough elements of value e to make the input size a power of two. The desired output is clearly the first N elements of the obtained output.

DFT [5, 6] is a transformation that takes as input an array $X[0..N-1]$ and computes the array $Y[0..N-1]$: $Y[k] = \sum_{l=0}^{N-1} X[k]e^{\frac{2\pi i}{N}kl}$. If N is not a power of 2, we can pad enough zeros to the input array X to make its size a power of 2 (2^K, say), and then apply FFT, obtaining an array Y' of size 2^K. The desired output $Y[0..N-1]$ is simply the first N elements of Y'.

Finally, sorting behaves similarly. If the size N of the input $X[0..N-1]$ to be sorted is not a power of two, we pad a certain number of $+\infty$ to X to raise its size to a power of two (2^K), then apply a parallelizable sorting algorithm such as bitonic sorting [3]. The desired output consists of the first N elements of the obtained output.

Translating such serial recursive algorithms to parallel SIMD code clearly reduces to the previous case (of full trees). The code translation algorithm first generates a SIMD target code exactly as in the previous subsection (for full trees), then adds one piece of code to the beginning of the SIMD target code, and another piece to the end. The first piece pads the input of the target code with an easily determinable number of copies of a certain value; that value may be automatically computed for cetain standard operations, or may be supplied by the user in an interactive program restructuring environment. The other piece of code is to recover the desired solution from the solution of the extended input. This recovery code may be generated automatically for certain algorithms by standard scalar data-flow analysis, or may be done interactively with the user in more complex situations.

3.3 Case 3: Non-Paddable Input

Recall that in this case the tree is not full, and the input I cannot be padded as in the previous case. Therefore, the structure of the tree varies depending on the input size (and possibly on the input values). This lack of *a priori* knowledge of the tree structure prevents us from knowing the masks and the communication patterns before run time of the supposed target code. In other terms, for every new input, the code translation algorithm has to be invoked to generate the masks and communication instructions, and integrate them to the previously generated computation instructions, effectively producing a new version of the SIMD target code for every new input.

This scenario is very slow for two reasons. First, the regeneration of a new target code for each new input slows down the execution. Second and more importantly, some of the levels of the tree for certain input sizes may be so irregular as to make the set of hosting processors unrepresentable by just one or even a small number of masks (or even by a small number of free-style if-then statements); rather, a large number of masks (or if-statements), sometimes linearly proportional to the number of involved processors, may be needed. The time to broadcast all these masks per instruction could make the SIMD code take longer time than the serial code that the SIMD code is supposed to speed up. Consequently, executing the tree in a top-down MIMD way, where a new process is spawned for every recursive call, avoids this problem and may yield better speedup. Although considerable work has been done on SIMD vs. MIMD computation, none of this work relates to recursive code translation, particularly to our problem at hand. Further work is then needed to characterize the serial recursive algorithms that are better suited for MIMD than for SIMD, and to study the tradeoffs.

4 Applications

In this section we apply our code generation algorithm to 2 representative recursive codes, namely, addition and FFT, each applied to N elements; N is assumed to be a power of 2 (if not, the input is padded appropriately to make N a power of 2 as we saw in the previous section). For brevity, we just derive the instruction sequences and their masks, without presenting the SIMD code of each of these algorithms, since the code is identical to that generated by SIMD-CODE-GENERATE of Figure 6. Each of these algorithms will be treated separately.

4.1 Addition

The recursive code for addition was presented earlier in Figure 2. Here is a brief but systematic analysis and development of the instruction/mask sequences.

- σ=2. $n = 2$. $N = \sigma \times n^l = 2^{l+1}$.
- Specified output layout: the final output is to be in processor 0.
- Analysis of the partition step
  - Canonical layout; no partition processing.
  - Therefore, each processor will have σ (=2) input elements: $a[2i]$ and $a[2i+1]$ assigned to processor i, for $i = 0, 1, \ldots, 2^l - 1$.
- Analysis of the basis step: A single instruction, namely, addition of the 2 local elements.
- Analysis of the merge step

- $CP_{merge}(2^l)$: one instruction, namely, addition of the subsolutions.
- $M^{CP}_{merge}(2^l)$: 0^l (because processor 0, whose binary ID is 0^l, was specified to hold the final output, and must thus compute it).
- Communication: No intra-merge communication because the merging is done by a single processor. As for inter-level communication, note that since the final output is to be put in processor 0, it follows that the output from each merge-kernel must be put in the 0-th processor of the machine part hosting that kernel. Consequently, the 0-th processor of the part hosting the right REC-node of *merge-kernel*(2^l) must send its local output to the 0-th processor of the part hosting the left REC-node of the merge-kernel. The 0-th processor of the right REC-node is 10^{l-1}, and that of the left REC-node is 0^l. Thus,
- $CM_{merge}(2^l)$: $10^{l-1} \rightarrow 0^l$, which is covered by the n-cube basic instruction e_l.
- M^{CM}_{merge}: 10^{l-1} (i.e., the ID of the sending processor).

- Conclusion
  - $Up.CP_k = CP_{merge}(2^k)$= one instruction, namely, addition of the subsolutions from the 2 children.
  - $Up.M^{CP}_k = *^{l-k} \bullet M^{CP}_{merge}(2^k) = *^{l-k}0^k$.
  - $Up.CM_k = CM_{merge}(2^k) = e_k$.
  - $Up.M^{CM}_k = *^{l-k} \bullet M^{CM}_{merge}(2^k) = *^{l-k}10^{k-1}$.
  - No intra-merge communication

4.2 FFT

The definition of the Discrete Fourier Transform (DFT) was reviewed earlier, and the recursive serial code for FFT is shown below.

```
FFT (input:X(0..N-1), output: Y(0..N-1))
begin
    if N = 1
        Y(0) = X(0);   {The basis step}
        return;
    endif

    { Now N > 1, and the input X will be partitioned
    into two arrays X'(0..N/2-1) and X''(0..N/2-1)}.
    for j = 0 to N/2-1
        X'(j) = X(2 × j);
        X''(j) = X(2 × j + 1);
    endfor

    {Next is recursion}
    FFT(X', U(0..N/2-1));
    FFT(X'', V(0..N/2-1);

    {Merge is next}
    for j = 0 to N/2-1
        Y(j) = U(j) + e^{(2πi/N) j} V(j);    {i = √-1}
        Y(j + N/2) = U(j) - e^{(2πi/N) j} V(j);
    endfor
end
```

As in the previous subsection, we present a brief but systematic analysis and development of the instruction/mask sequences.

- σ=1. $n = 2$. $N = \sigma \times n^l = 2^l$.
- Layout of the final output: $Y(j)$ in processor j, for $k = 0, 1, \ldots, 2^l - 1$.
- Analysis of the partition step
  - The partitioning is simply a reordering of the input before it is split into 2 equal halves X' and X''.
  - Therefore, no $CP_{partition}$ and no intra-partition communication.
  - Input layout: Canonical, as stipulated in the previous section. that is, $X(j)$ is assigned to processor j, for $j = 0, 1, \ldots, 2^l - 1$.
  - Inter-level communication: Under the same stipulation, $X'(j)$ must be in the j-th processor of the first half of the machine, and $X''(j)$ must be in the j-th processor of the second half of the machine. That is, $X'(j)$ is in processor j, and $X''(j)$ is in processor $j + 2^{l-1}$. Therefore,
  - $CM_{partition}(2^l)$: $2j \rightarrow j$, and $2j + 1 \rightarrow j + 2^{l-1}$, for $j = 0, 1, \ldots, 2^{l-1} - 1$. This happens to be the perfect unshuffle U_l.
  - $M^{CM}_{partition}$: All the processors, i.e., $*^l$.
- Analysis of the basis step: A single instruction, namely, assignment of the input element to the output element.
- Analysis of the merge step
  - $CP_{merge}(2^l)$: 3 instructions $\{i_1, i_2, i_3\}$. i_1 is the multiplication of $e^{\frac{2\pi i}{N}j}$ by the local data output $V(j)$, for all $j = 0, 1, \ldots, 2^{l-1} - 1$, by processors $j + 2^{l-1}$. i_2 is one addition to compute $Y(j)$ by processor j for all $j = 0, 1, \ldots, 2^{l-1} - 1$. i_3 is one subtraction to compute $Y(j + 2^{l-1})$ by processor $j + 2^{l-1}$ for all $j = 0, 1, \ldots, 2^{l-1} - 1$.
  - M^{CP}_{merge}: $1*^{l-1}$ for i_1; $0*^{l-1}$ for i_2; and $1*^{l-1}$ for i_3.
  - Inter-level communication: For $j = 0, 1, \ldots, 2^{l-1} - 1$, processor j needs $e^{\frac{2\pi i}{N}j}V(j)$ from processor $j + 2^{l-1}$, and the latter processor needs $U(j)$ from the former processor. Hence, one single communication instruction is needed, namely, the n-cube e_{l-1}, to be performed by all the processors. That is,
  - $CM_{merge}(2^l)$: e_{l-1}.
  - $M^{CM}_{merge}(2^l)$: $*^l$.
  - Intra-merge communication: None.
- Conclusion
  - No partition computation
  - $Down.CM_k = the\ unshuffle\ U_k$.
  - $Down.M^{CM}_k = *^{l-k}*^k = *^l$;
  - $Up.CP_k = \{i_1, i_2, i_3\}$, defined above.

- $Up.M_k^{CP} = \{*^{l-k}1*^{k-1}, *^{l-k}0*^{k-1}, *^{l-k}1*^{k-1}\}$.
- $Up.CM_k = e_{k-1}$.
- $Up.M_k^{CM} = *^{l-k}*^k = *^l$.
- No intra-merge communication

5 Conclusions and Future Work

In this paper we have developed an approach and an efficient translation algorithm that translates recursive serial codes of a common recursive structure to SIMD target codes. The translation works very well on balanced recursive codes where the input can be padded to an appropriate size (e.g., power of 2). The translation algorithm was applied to semigroup computations and FFT to illustrate its working and its power. It was also argued that for recursive codes where the input size is not a desired value and the input cannot be padded appropriately, the SIMD mode of execution is too slow, and the MIMD mode is preferable.

Future work should focus on the tradeoff of SIMD vs. MIMD execution of recursive codes whose input cannot be appropriately padded to a desired size. The tradeoff study is to characterize further this class of codes and determine more closely when the MIMD mode is preferable, and possibly to automate this arbitration. Another extension of the work is to generalize the approach to other recursive structures such as interleaved recursion and irregular recursion (e.g., when partitioning itself uses recursion).

References

[1] J. R. Allen and K. Kennedy, "Automatic Loop Interchange," *Proc. SIGPLAN '84 Symp. on Compiler Construction,* Montreal, Canada, pp. 233–246, June 1984.

[2] U. Banerjee, *Dependence Analysis for Supercomputing,* Kluwer Academic Publishers, Norwell, Mass., 1988.

[3] K. E. Batcher, "Sorting Networks and their Applications," *1968 Spring Joint Comput. Conf., AFIPS Conf.* Vol. 32, Washington, D.C.: Thompson, 1968, pp. 307–314.

[4] V. E. Benes, *Mathematical Theory on Connecting Networks and Telephone Traffic,* Academic Press, New York, 1965.

[5] J. W. Cooley and J. W. Tuckey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. of Comput.,* Vol. 19, pp. 297–301.

[6] R. Gonzales and R. Woods, *Digital Image Processing,* (Chapter 3) Addison-Wesley, 1992.

[7] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic Dependence Analysis for High-Performance Parallelizing Compilers," in *Advances in Languages and Compilers for Parallel Processing,* A. Nicolau, A. Gelertner, T. Gross, and D. Padua (Eds.), MIT Press, 1991.

[8] D. K. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comput.*, C-24, pp. 1145–155, Dec. 1975.

[9] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* MIT Press,1989.

[10] M. Wolfe, and U. Banerjee, "Data Dependence and its Application to Parallel Processing," *Int'l Journal of Parallel Programming,* 16(2), pp. 137–178, April 1987.

[11] M. Wolfe, and M. Lam, "An algorithmic Approach to Compound Loop Transformations," in *Advances in Languages and Compilers for Parallel Processing,* A. Nicolau et al (Eds.), MIT Press, 1991.